Bipartite Drawing with Minimum Edge Crossings of Binary Trees and 3-Cayley Trees

Eliezer A. Albacea
Institute of Computer Science
University of the Philippines Los Baños
College, Laguna

eaalbacea@uplb.edu.ph

ABSTRACT

In this paper, we present a simple algorithm for bipartite drawing with minimum edge crossings of binary trees that has a running time of *O(n)*. Together with the drawing, the algorithm also computes the bipartite crossing numbers of binary trees using the same amount of time. The algorithm for binary trees were used to produce a bipartite drawing and used to compute the bipartite crossing numbers of 3-Cayley trees. The algorithm produced also runs in *O(n)*. For the case of trees, the computation of bipartite crossing numbers has been shown by Shahrokhi, et.al. [9] to be computable in $O(n^{1.6})$ time. Hence, our result improves that of Shahrokhi et al. [9] for the case of binary trees.

Keywords

Bipartite drawing, bipartite crossing number, graph drawing, binary tree, 3-Cayley tree.

1. INTRODUCTION

Let $G = (V_0, V_1, E)$ be a connected bipartite graph, where V_0, V_1 is the bipartition of vertices into two independent sets. A bipartite drawing of G consists of placing the vertices V_0 and V_1 into distinct points on two parallel lines x and y, respectively, and then drawing each edge with one straight line segment which connects the points of x and y where the endvertices of the edge where placed. It is unavoidable in some cases that these edges cross. One of the aesthetic criteria for graph drawing is one with minimum edge crossings.

The bipartite crossing number of G, $bcr(G)$, is the minimum number of crossings of edges over all bipartite drawing of G. The problem of bipartite drawing with minimum edge crossing is related to the problem of computing the bipartite crossing number. The problem of finding the bipartite crossing number was first studied by Harary [5] and Harary and Schwenk [6] and independently proposed by Watkins [12] as cited in Shahrokhi, et.al. [8]. The bipartite crossing number problem was shown to be NP-complete by Garey and Johnson [4]. However, it was shown to be solvable in polynomial time for bipartite permutation graphs by Spinrad, et. al. [10] and for trees by Shahrokhi, et. al. [9].

Most efforts in bipartite drawing has been on minimizing the edge crossings. First, there is the problem of one-sided crossing minimization where one side is fixed and then the other side is permuted so as to minimize the number of edge crossings. The most popular heuristics for this problem is the barycenter heuristic of Sugiyama, et.al. [11], the median heuristic of Eades and Wormald [3], the split heuristic of Eades and Kelly [2], the sifting heuristic of Matuszewki, et.al. [7] and many other heuristics. Although these heuristics attempt to minimize the number of edge crossings, all these methods, however, will not guarantee the production of the optimum solution or drawing exhibiting the bipartite crossing number.

Our objective in this paper is to produce a drawing with minimum edge crossings. Albacea and Tabadda [1] have shown a bipartite drawing with minimum edge crossings of complete binary trees. In this paper, we show an algorithm that produces a drawing with minimum edge crossings for binary trees and 3-Cayley trees. The algorithms for binary trees and 3-Cayley trees run in *O(n)*. As a consequence of the drawing algorithm, we also obtain an algorithm for computing the bipartite crossing numbers of binary trees and 3-Cayley trees.

2. NOTATIONS AND DEFINITIONS

A *binary tree* $T = (V,E)$ is a tree in which each node has at most 2 children. It is usually rooted, we assume that the root is vertex r. Assume that the root has level equal to 0 and that the binary tree has height (maximum level) h. Each node v in the binary tree is assigned a weight, *weight(v)*, equal to the number of edges in the subtree rooted at v. If a vertex has two children, then one of them is called the right child and the other is called the left child.

A tree in which each non-leaf node has a constant number of branches n is called an *n-Cayley tree*. Hence, a *3-Cayley tree* has three branches in each of its internal nodes and has one branch each in its leaf nodes.

3. THE ALGORITHM

3.1 Description

Let a subtree be rooted at vertex v and denoted T_v. The vertex v has vl and vr as left and right children, respectively. Let vll and vlr be the left child and right child of vl, respectively, and let vrl and vrr be the left child and right child of vr, respectively. Let crossnumber(v) be the minimum number of crossings of edges in T_v plus the edge connecting v and its parent. Hence, we assume that v is not the root node. Let us consider a certain configuration of T_v as given in Figure 1.

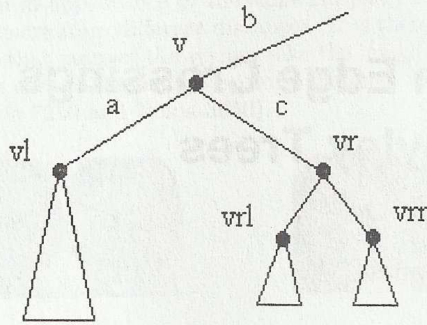

Figure 1. A subtree rooted at v

Suppose v is a left child. Suppose a bipartite drawing has already been drawn for T_{vl}, T_{vrl} and T_{vrr}. Adding v and vr to the bipartite drawing will produce an additional crossings which is minimal if we draw edge (vr, vrl) and the edges in T_{vrl} crossing the edge *a* in Figure 1 and the edge (vr, vrr) and the edges in T_{vrr} crossing the edge *b* in Figure 1. All other drawings will produce more edge crossings. Hence, the crossing number in T_v has minimal additional number of crossing and this number of crossing is given by the formula:

crossnumber(v) = crossnumber(vl) + weight(vr)
+ crossnumber(vrl) + crossnumber (vrr).

Similarly, if v is a right child the corresponding formula is:

crossnumber(v) = crossnumber(vr) + weight(vl)
+ crossnumber(vll) + crossnumber (vlr).

An example is shown in Figure 2 illustrating the computation of crossing number using the formula above.

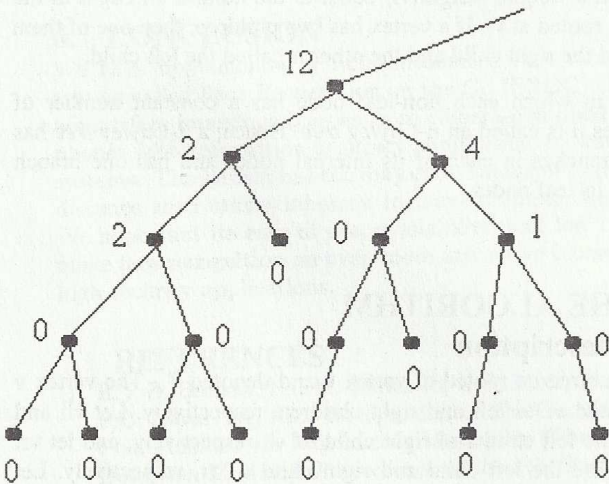

Figure 2. Illustration of crossing number computation.

The algorithm that we will present will be using a twisting procedure that simply produces the mirror image of a subtree. For illustration of the twisting procedure see Figure 3.

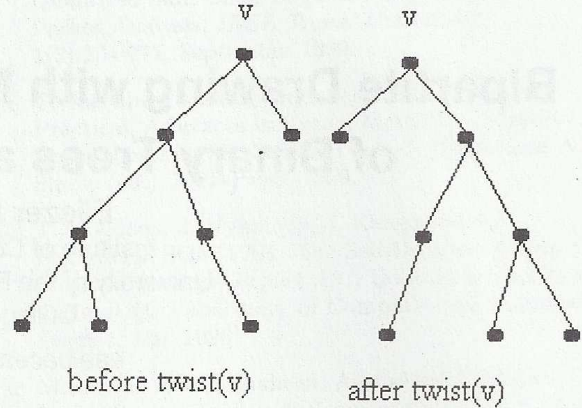

Figure 3. Before and after twisting.

It is obvious that twisting a subtree does not affect the crossing numbers of the subtrees rooted at the left and right children of the root of the subtree. For example, consider Figure 1. Twisting at vertex v will not change the number of edge crossings in the subtrees T_{vl} and T_{vr}. However, it may change the number of edge crossings in T_v.

Now, we can outline the algorithm as follows:

Algorithm: BinaryTreetoBipartite

1. Compute the weight(v) for each v ∈ V.
2. For each vertex v at level h to level 1 do

   case weight(v)

   0: crossnumber(v) = 0

   1: crossnumber(v) = 0

   if v is a right child then make the single child of v a right child

   if v is a left child then make the single child of v a left child

   2: crossnumber(v) = 0

   >2: if v is a left child

   u = crossnumber(vl) + weight(vr)
   + crossnumber(vrl) + crossnumber(vrr)

   t = crossnumber(vr) + weight(vl)
   + crossnumber(vll) + crossnumber(vlr)

   if u > t then crossnumber(v) = t
   twist(v)

   if u < t then crossnumber(v) = u

   if u = t then
   if weight(vr) > weight(vl) then twist(v)

   if v is a right child

   u =crossnumber(vr) + weight(vl)
   + crossnumber(vll) + crossnumber(vlr)

   t = crossnumber(vl) + weight(vr)

+ crossnumber(vrl) + crossnumber(vrr)

if u > t then crossnumber(v) = t

twist(v)

if u < t then crossnumber(v) = u

if u = t then

if weight(vl) > weight(vr) then twist(v)

3. crossumber(r) = crossnumber(rl) + crossnumber(rr), where r is the root of the tree.

4. Call the drawing algorithm DrawBipartite.

Lemma 1. The algorithm computes the minimum number of edge crossings.

Proof: The inductive method can be used to prove the lemma. Note that for subtrees of heights 1, the algorithm produces minimum edge crossings.

Next, consider a subtree as illustrated in Figure 1. Assuming that the subtrees T_l, T_r, T_{rl} and T_{rr} have minimum edge crossings, the algorithm computes the minimum edge crossing at T_v considering the original subtree and its mirror image by twisting. Hence, the algorithm assures that it has minimal edge crossings at T_v after Step (2). By induction, therefore, the minimum number of edge crossings is computed all the way to the root.

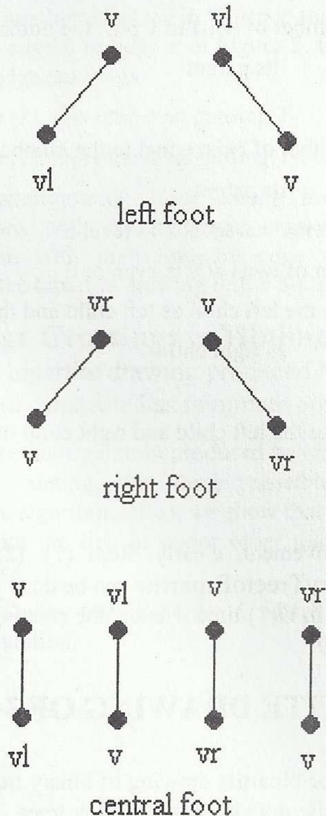

Figure 4. Three types of feet.

Next, we outline the drawing algorithm. Given a node v and its two possible children vl and vr. The child vl may be drawn at the left of v in which case we call vertex vl as a left foot taking a left position and the edge (v, vl) as a left leg. Similarly, the child vr may be drawn at the right of v in which case we call vertex vr as a right foot taking a right position and the edge (v, vr) as a right leg. Finally, vl and vr may be drawn in the same vertical line with v in which case vl or vr is a central foot taking a central position and the edge (v, vl) or edge (v, vr) is a central leg. See Figure 4 to illustrate the different configuration of the edges in the bipartite drawing.

Now, we outline the rules for drawing the legs. This is obvious from algorithm DrawLegs.

Algorithm: DrawLegs(v)

1. switch(position of v)

left:

if vr exists then make vr a central foot

if vl exists then make vl a left foot

central:

if vr exists then make vr a right foot

if vl exists then make vl a left foot

right:

if vr exists then make vr a right foot

if vl exists then make vl a central foot

All the legs can be drawn as follows:

Algorithm: DrawBipartite

1. Make the left child of the root a left foot and the right child a right foot.

2. For each vertex v in level 1 to level h-1

DrawLegs(v)

It should be noted that algorithm DrawBipartite draws the edges following the rules we adopted in our computation of the number of edge crossings. Take Figure 1 for example. Assuming v is a left foot, using DrawLegs vl will also be a left foot and vr is a central foot. Drawing further vrl will be a left foot, thus edge (v, vrl) will cross edge *a* in Figure 1 and the rest of the edges in T_{vl} will also cross edge *a* in Figure 1. On the other hand, vr will be a right foot and edge (v, vr) will cross edge *b* in Figure 1 and so are the edges in T_{vr}.

3.2 Analysis of the Algorithm

Assuming a binary tree with n vertices. In algorithm BinarytoBipartite, Step (1) can be done in *O(n)*. Step (2) involves the procedure twist(v) which has a cost of *O(weight(v))*. Hence, the total cost is

$$O\left(\sum_{v\in V-\{r\}} \text{weight}(v)\right)$$

It is $v \in V - \{r\}$ because Step (2) excludes the root r of the tree. The best case will be obtained when the binary tree is a complete binary tree, where

$$O\left(\sum_{v \in V-\{r\}} \text{weight}(v)\right) = \sum_{i=1}^{h-1}(2^{h-i+1}-1)2^i = \left(2^{h+1}\right)h+2-3\left(2^h\right)$$

In a complete binary tree, $h = \log n$. Hence, a total cost for a complete binary tree of $O(n \log n)$ time. The worst case, however, occurs when the tree is a unary tree where

$$O\left(\sum_{v \in V-\{r\}} \text{weight}(v)\right) = O(n^2)$$

Hence, the running time of Step (2) will range from $O(n \log n)$ to $O(n^2)$. Step (3) *is* $O(1)$ while Step (4) is $O(n)$ time. Hence, we have a total running time that ranges from $O(n \log n)$) to $O(n^2)$.

3.3 Improving the Running Time

One would notice that the running time of the algorithm is dominated by the twist operation. We outline an improvement where we do not do the twist immediately but rather we postpone the twisting until after the bipartite crossing number has been computed. This algorithm is given below.

Algorithm: ImprovedBinaryTreetoBipartite

1. Compute the weight(v) for each $v \in V$ and mark each $v \in V$ as untwisted.
2. For each vertex v at level h to level 1 do

    case weight(v)

        0: crossnumber(v) = 0

        1: crossnumber(v) = 0

            if v is a right child then make the single child of v a right child

            if v is a left child then make the single child of v a left child

        2: crossnumber(v) = 0

        >2: if v is a left child

            u = crossnumber(vl) + weight(vr) + crossnumber(vrl) + crossnumber(vrr)

            t = crossnumber(vr) + weight(vl) + crossnumber(vll) + crossnumber(vlr)

            if u > t then crossnumber(v) = t

                mark v as twisted

            if u < t then crossnumber(v) = u

            if u = t then

                if weight(vr) > weight(vl) then mark v as twisted

        if v is a right child

            u =crossnumber(vr) + weight(vl) + crossnumber(vll) + crossnumber(vlr)

            t = crossnumber(vl) + weight(vr) + crossnumber(vrl) + crossnumber(vrr)

            if u > t then crossnumber(v) = t

                mark v as twisted

            if u < t then crossnumber(v) = u

            if u = t then

                if weight(vl) > weight(vr) then mark v as twisted

3. crossumber(r) = crossnumber(rl) + crossnumber(rr), where r is the root of the tree.
4. Call CarryOutTwisting
5. Call the drawing algorithm DrawBipartite.

The idea of the algorithm CarryOutTwisting is that twisting is done only on the children of a node. The rest of the subtree is twisted depending on whether they need to be twisted on not. The first step of the algorithm is to count the number of times the subtree rooted at vertex needs to be twisted.

Algorithm: CarryOutTwisting

1. Mark the root as untwisted and set number of twist of the root to 0.
2. For each vertex v at level 1 to level h

    if vertex v is marked twisted

        set number of twist to 1 plus the number of twist of its parent

    else

        set number of twist equal to the number of twist of its parent.

3. For each vertex v at level 1 to level h

    if number of twist at v is even or 0

        retain the left child as left child and the right child as right child

    else

        reverse the left child and right child of v

With this improvement, clearly Steps (1), (2), (4) and (5) of ImprovedBinaryTreetoBipartite can be done in O(n) time, Step (3) can be done in O(1) time. Hence, the running of the improved algorithm is $O(n)$.

4. BIPARTITE DRAWING OF 3-CAYLEY TREES

The algorithm for bipartite drawing of binary trees can be used to produce a bipartite drawing for 3-Cayley trees. The outline of the algorithm for bipartite drawing of 3-Cayley trees is given in the next subsection.

4.1 The Algorithm

The outline of the algorithm is as follows:

Algorithm: 3-CayleyTree_to_Bipartite

1. Root the 3-Cayley tree at its center r. This will produce a tree with configuration shown in Figure 5,

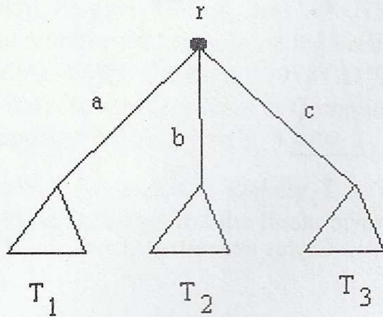

Figure 5. A 3-Cayley rooted at its center.

where T_1, T_2 and T_3 are binary trees.

2. Disregarding T_1, compute the bipartite crossing number of the binary tree formed from r, T_2 and T_3. Compute the bipartite crossing number of T_1. Then, position T_1 in between T_2 and T_3. When T_1 is positioned in between T_2 and T_3, this will have a crossing number equal to the crossing number of T_1 plus the number of edges in T_1 since the edges in T_1 will cross either edge b or edge c in Figure 5. Compute the total number of edge crossings.
3. Repeat Step (2), this time disregarding T_2.
4. Repeat Step (2), this time disregarding T_3.
5. Find the configuration from among the three possible configurations with the minimum crossing number. The configuration with the minimum edge crossing will be retained as the bipartite drawing of the 3-Cayley tree.

4.2 The Edge Crossings is Minimum

Lemma 2. The bipartite drawing produced by the algorithm 3-CayleyTree_to_Bipartite has minimum edge crossings.

Proof: Clearly, the configuration produced has the minimum edge crossings from among the three possible configurations considered by the algorithm. Next, we show that it stays minimum even if we re-root the tree in nodes other than the center. We consider two cases.

Consider the 3-Cayley tree in Figure 6 where the figure is the minimum configuration.

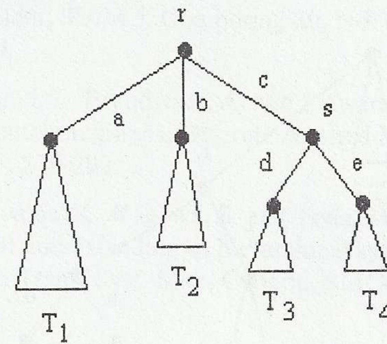

Figure 6. A 3-Cayley tree rooted at the center r.

Case 1: Re-root the tree at a node in the leftmost or rightmost subtrees, i.e., at a node in T_1 or node in the subtree formed by s, T_3 and T_4.

We consider the case where it is re-rooted at the rightmost binary tree. If the tree in Figure 6. is re-rooted at a node found in the rightmost binary tree, i.e. re-rooted at node s, we obtain the tree in Figure 7.

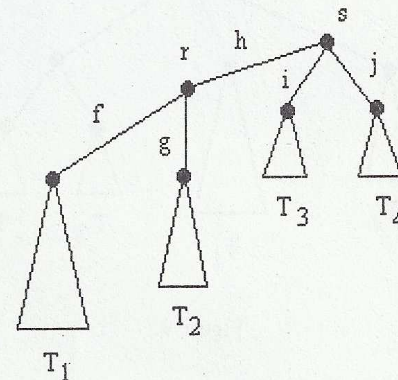

Figure 7. A 3-Cayley tree re-rooted at node s.

When the tree is re-rooted, we note that the number of edge crossings in T_1 and T_4 will not change. Next, let us consider the number of crossings in T_2. In Figure 6, the edges in T_2 will cross either edge a or edge c. Note that when the tree is re-rooted, the edges in T_2 will either cross edge f or edge h. Hence, its number of crossings will not be affected when the tree is re-rooted to s. It is similar with T_3, where originally it will cross either edge c or edge e and in the re-rooted tree it will cross either edge h or edge j. Hence, it too will not change its number of crossings. The same argument will apply if it is re-rooted again in another node found in T_4.

Case 2: Re-root the tree at the middle subtree. When the tree is re-rooted at the middle subtree we obtain one possible configuration given in Figure 8.

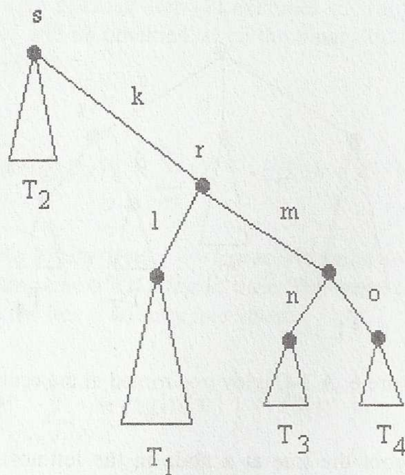

Figure 8. A 3-Cayley tree re-rooted at the middle binary tree.

Using the arguments in Case 1, the configuration in Figure 8 will have the same number of edge crossing as in the tree given in Figure 9.

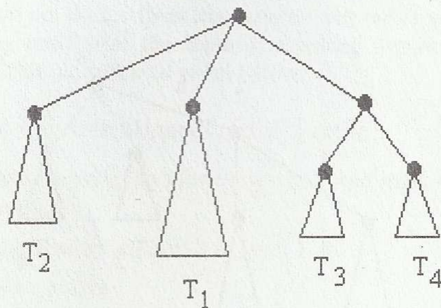

Figure 9.

However, this configuration has been shown to have a crossing number greater than the minimum configuration in Figure 6 by virtue of the fact that that the configuration in Figure 6 has the minimum number of edge crossings. Hence, the number of edge crossings in a tree in Figure 8 has greater number of edge crossings than the tree in Figure 6.

The same argument applies to the other possible configuration, i.e. configuration given in Figure 10.

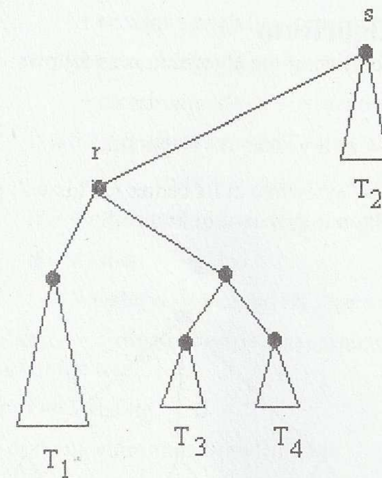

Figure 10. A 3-Cayley tree re-rooted at the middle binary tree.

4.3 The Running Time

The size of the middle binary tree may range from 1 to about $n/3$. In which case the total size of the remaining two binary trees will range from about $2n/3$ to $n-1$. Hence, the algorithm will run in $O(n)$.

5. CONCLUSIONS

We have presented a simple algorithm for bipartite drawing of a binary tree. The algorithm also produced a drawing with minimum edge crossings and it computes this minimum number of edge crossings. The algorithm has a running time of $O(n)$. This improves the $O(n^{1.6})$ time algorithm of Shahrokhi, et.al. [9] for the computation of bipartite crossing numbers when the tree is a binary tree. Also presented is an algorithm for bipartite drawing and computation of bipartite crossing numbers of 3-Cayley trees.

6. REFERENCES

[1] Albacea, E.A. and Tabbada, X.A.Q. A linear algorithm for bipartite drawing of complete binary trees. 3rd Philippine Computing Science Congress, Philippine Science High School, February 8-9, 2003.

[2] Eades, P. and Kelly, D. Heuristics for drawing 2-layered networks, Ars Combinatorica 21-A, 1986, 89-98.

[3] Eades, P. and Wormald, N. Edge crossings in drawings of bipartite graphs, Algorithmica 11, 1994, 379-403.

[4] Garey, M.R. and Johnson, D.S. Crossing number is NP-complete, SIAM J. Algebraic and Discrete Methods 4, 1983, 312-316.

[5] Harary, F. Determinants, permanents and bipartite graphs, Mathematical Magazine 42, 1969, 146-148.

[6] Harary, F. and Schwenk, A. A new crossing number for bipartite graphs, Utilitas Mathematica 1, 1972, 203-209.

[7] Matuszewki, C., Shonfeld, R., and Molitor, P. Using sifting for k-layer straightline crossing minimization, Proceedings Graph Drawing 1999 (GD '99), 1999, 217-224.

[8] Shahrokhi, F., Szekely, L.A. and Vrto, I. Bipartite crossing numbers of meshes and hypercubes, Proceedings Graph Drawing 1997 (GD '97), 1997, Rome, Italy, in Lecture Notes in Computer Science 1353, Springer-Verlag, Berlin, 37-46.

[9] Shahrokhi, F., Sykora, O., Szekely, L.A. and Vrto, I. On bipartite drawings and the linear arrangement problem, SIAM J. Computing 30, No. 6, 2001, 1773-1789.

[10] Spinrad, J., Brandstadt, A. and Stewart, L. Bipartite permutation graphs, Discrete Applied Mathematics 19, 1987, 279-292.

[11] Sugiyama, K., Tagawa, S. and Toda, M. Methods for visual understanding of hierarchical system structures, IEEE Trans. Syst. Man, Cybern., SMC-11, 1981, 109-125.

[12] Watkins, M.E. A special crossing number for bipartite graphs: a research problem, Annals of New York Academy of Sciences 175, 1970, 405-410.